Wer rastet, der rostet

Ein Rust-Kurs für Anfänger

Hendrik Wolff

Anton Obersteiner

Sommersemester 2023

Funktionen

```
fn spass_mit_funktionen() {
    let right = 17;
    let light = seven();
    println!("adding {left} and {right}: {}", add(left, right));
}
fn add(l: u8, r: u8) -> u8 {
    return left + right;
}
fn seven() -> u8 {
    7 // implicit return
}
```

Funktionen

Nutzt aussagekräftige Variablennamen!

```
fn spass_mit_funktionen() {
    let right = 17;
    let left = seven();
    println!("adding {left} and {right}: {}", add(left, right));
}
fn add(left: u8, right: u8) -> u8 {
    return left + right;
}
fn seven() -> u8 {
    7 // implicit return
}
```

Funktionen

Nutzt aussagekräftige Variablennamen!

```
fn spass_mit_funktionen() {
    let right = 17;
    let left = seven();
    println!("adding {left} and {right}: {}", add(left, right));
}
fn add(left: u8, right: u8) -> u8 {
    return left + right;
}
fn seven() -> u8 {
    7 // implicit return
}

let closure = |param1, param2| { /* function body*/ };
```

Quiz 3: return

Was wird passieren, wenn diese funktionen aufgerufen werden?

```
fn add_two_a(a: u32) -> u32 {
    return a + 2;
}
fn add_two_b(b: u32) -> u32 {
    b + 2;
}
fn add_two_c(c: u32) -> u32 {
    c + 2
}
```

Quiz 3: return

Was wird passieren, wenn diese funktionen aufgerufen werden?

```
fn add_two_a(a: u32) -> u32 {
    return a + 2;
}
fn add_two_b(b: u32) -> u32 {
    b + 2;
}
fn add_two_c(c: u32) -> u32 {
    c + 2
}
```

⇒ compilation error!

Comments

```
// This is a comment.  Multi-line comments
// generally are written this way.

/* You can use this style of comment too. */

/// This is a doc comment.
```

Statements und Expressions

Statements:

Instruktionen ohne return Wert

```
let a = 2; /*
^^^^^^^^^^ statement */
println!("Hello students!"); /*
^^^^^^^^^^^^^^^^^^^^^^^^^^^ statement */
fn goodbye() -> () {
    println!("Goodbye");
}
```

Statements und Expressions

Statements:

Instruktionen ohne return Wert

```
let a = 2; /*
^^^^^^^^^^ statement */
println!("Hello students!"); /*
^^^^^^^^^^^^^^^^^^^^^^^^^^^ statement */
fn goodbye() -> () {
    println!("Goodbye");
}

// `-> ()` is implicit
fn goodbye() /* no return */ {
    println!("Goodbye");
}
```

Statements und Expressions

Statements:
Instruktionen ohne return Wert

```
let a = 2; /*
^^^^^^^^^^ statement */
println!("Hello students!"); /*
^^^^^^^^^^^^^^^^^^^^^^^^^^^ statement */
fn goodbye() -> () {
    println!("Goodbye");
}

// `-> ()` is implicit
fn goodbye() /* no return */ {
    println!("Goodbye");
}
```

Expressions:
evaluieren zu einem Wert

```
fn add_one(a: i32) -> i32 {
    a + 1
//  ^^^^^ expression
}
let n = add_one(2);
/*      ^^^^^^^^^^ expression
^^^^^^^^^^^^^^^^^^ statement
*/
```

Statements und Expressions

```
let a = 3;
let b = {
    let a = a + 1;
    a * 4
};
```

Statements und Expressions

```
let a = 3;
let b = {
    let a = a + 1;
    a * 4
};

let c = if a > 3 { a - 2 } else { a + 4 }; /*
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        |____ expression
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|____ statement */
```

Control flow - branching mit if und else

```
if students < 10 {
    println!("Kein Rust Kurs :(");
} else if students > 30 {
    println!("Raum überfüllt!!");
} else {
    println!("Rust Kurs findet statt :D");
}
```

Übung 2: Fibonacci

Berechne die `n`-te Fibonacci-Zahl mit Hilfe einer rekursiven Funktion.
Verwende dabei `u16` und ein implicit `return`.
Zusatz: Verwende gar kein `return`.

Gib deiner Fibonacci-Funktion Zahlen zwischen 0 bis 50.
Starte dein Programm mit `cargo run`. Was passiert und warum?

Übung 2: Fibonacci

Berechne die `n`-te Fibonacci-Zahl mit Hilfe einer rekursiven Funktion.
Verwende dabei `u16` und ein implicit `return`.
Zusatz: Verwende gar kein `return`.

Gib deiner Fibonacci-Funktion Zahlen zwischen 0 bis 50.
Starte dein Programm mit `cargo run`. Was passiert und warum?

```
fn fib(n: u16) -> u16 {
    if n <= 1 {
        return 1;
    }
    fib(n - 1) + fib(n - 2)
}
```

Übung 2: Fibonacci

Berechne die `n`-te Fibonacci-Zahl mit Hilfe einer rekursiven Funktion.
Verwende dabei `u16` und ein implicit `return`.
Zusatz: Verwende gar kein `return`.

Gib deiner Fibonacci-Funktion Zahlen zwischen 0 bis 50.
Starte dein Programm mit `cargo run`. Was passiert und warum?

```rust
fn fib(n: u16) -> u16 {
    if n <= 1 {
        return 1;
    }
    fib(n - 1) + fib(n - 2)
}
```

```rust
fn fib_no_return(n: u16) -> u16 {
    if n <= 1 {
        1
    } else {
        fib_no_return(n - 1) + fib_no_return(n - 2)
    }
}
```

Control flow - loops

Endlosschleife

```
loop {
    println!("HEYYEYAAEYAAAEYAEYAA");
}
```

Control flow - loops

Endlosschleife

```
loop {
    println!("HEYYEYAAEYAAAEYAEYAA");
}
```

```
let mut counter = 0;
let n = loop {
    counter += 1;
    if counter % 2 == 1 {
        println!("yes");
    } else if counter % 4 == 3 {
        continue;
    } else if counter > 20 {
        break counter + 2;
    }
}
```

Control flow - loops

Endlosschleife

```
loop {
    println!("HEYYEYAAEYAAAEYAEYAA");
}
```

```
let mut counter = 0;
let n = loop {
    counter += 1;
    if counter % 2 == 1 {
        println!("yes");
    } else if counter % 4 == 3 {
        continue;
    } else if counter > 20 {
        break counter + 2;
    }
}
```

Konditionalschleife

```
let mut counter = 5;
while counter >= 0 {
    counter -= 1;
    println!("no");
}
```

Wieviele “no" gibt das aus?

Control flow - loops

Endlosschleife

```
loop {
    println!("HEYYEYAAEYAAAEYAEYAA");
}
```

```
let mut counter = 0;
let n = loop {
    counter += 1;
    if counter % 2 == 1 {
        println!("yes");
    } else if counter % 4 == 3 {
        continue;
    } else if counter > 20 {
        break counter + 2;
    }
}
```

Konditionalschleife

```
let mut counter = 5;
while counter >= 0 {
    counter -= 1;
    println!("no");
}
```

Wieviele “no" gibt das aus?

⇒ 6

Control flow - loops

Über Elemente eines arrays iterieren.

`while`-Schleife

```
let arr = [1, 2, 3, 4, 5];
let mut idx = 0;
while idx < arr.len() {
    println!("{}", arr[idx]);
    index += 1;
}
```

Control flow - loops

Über Elemente eines arrays iterieren.

`while`-Schleife

```
let arr = [1, 2, 3, 4, 5];
let mut idx = 0;
while idx < arr.len() {
    println!("{}", arr[idx]);
    index += 1;
}
```

`for`-Schleife

```
let arr = [1, 2, 3, 4, 5];
for element in arr {
    println!("{element}");
}
```

Quiz 4: Control flow - labeled loops

Labeled loops

```
let mut counter = 0;
let x = 'outer: loop {
    'inner: loop {
        counter += 2;
        if counter > 2 {
            break 'outer counter * 4;
        } else {
            counter -= 1;
        }
    }
}
```

Quiz 4: Control flow - labeled loops

Labeled loops

```
let mut counter = 0;
let x = 'outer: loop {
    'inner: loop {
        counter += 2;
        if counter > 2 {
            break 'outer counter * 4;
        } else {
            counter -= 1;
        }
    }
}
```

Welchen Wert hat x am Ende?

Quiz 4: Control flow - labeled loops

Labeled loops

```
let mut counter = 0;
let x = 'outer: loop {
    'inner: loop {
        counter += 2;
        if counter > 2 {
            break 'outer counter * 4;
        } else {
            counter -= 1;
        }
    }
}
```

Welchen Wert hat x am Ende?

⇒ 12

Control flow - loops

Über eine range von Zahlen iterieren.

rechts-offene range

```
for n in 1..4 {
    println!("{n}");
}
```

$[1, 4) \Rightarrow 1, 2, 3$

Control flow - loops

Über eine range von Zahlen iterieren.

rechts-offene range

```
for n in 1..4 {
    println!("{n}");
}
```

$[1,4) \Rightarrow 1, 2, 3$

rechts-geschlossene range

```
for n in 1..=4 {
    println!("{n}");
}
```

$[1,4] \Rightarrow 1, 2, 3, 4$